**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по лабораторной работе № 3**

**по дисциплине «Алгоритмы и структуры данных»**

**Вариант №3**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8302 | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | Никулин Л.А. |
| Преподаватель | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | Тутуева А.В. |

**1.Цель работы**

Реализовать программу принимающую список рейсов и цены за прямой и обратный и рейс и, в которой пользователь вводит город отправления и назначения и получает самый выгодный рейс или получает информацию о невозможности совершения перелетов методом Флойда-Уоршелла.

**2.Описание реализуемого класса и методов**

|  |  |
| --- | --- |
| Matrix | Двумерный массив цен на рейсы, схожий с матрицей смежности |
| int size\_of\_matrix | Размер матрицы смежности |
| Map<string, int>\* map\_City\_name\_to\_index | Для хранения названия и получения его индекса |
| Map<int, string>\* map\_index\_to\_name\_City | Для хранения индекса и получения его названия города |

**3.Оценка временной сложности алгоритмов**

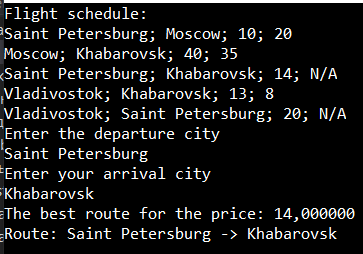
|  |  |
| --- | --- |
| *string getResult(string start\_City, string end\_City)* | O(N^3) |
| *get\_list\_symbol()* | O(1) |
| *print\_path (int i, int j, int\*\* p, Map<int, string>\* map\_index\_to\_name\_City, string&cur)* | O(N^2) |

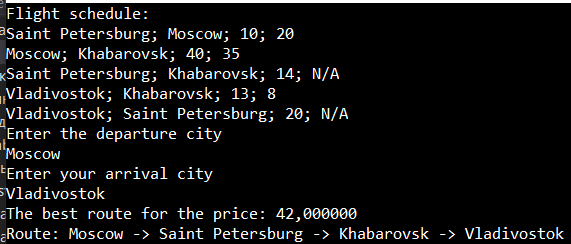
**4.Описание реализованных unit-тестов**

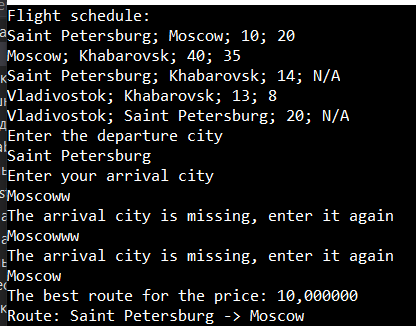
Реализованные мною тесты проверяют правильное нахождение выгодного перелёта. Я рассмотрел две ситуации когда перелёт возможен и когда нет.

|  |  |
| --- | --- |
| Test\_Path\_is\_avaible | Тест, проверяющий ситуацию, когда перелёт возможен |
| Test\_Path\_is\_not\_avaible | Тест, проверяющий ситуацию, когда перелёт невозможен |

**5.Пример работы программы**

****

****

****

**Листинг**

**Lab3.cpp:**

#include <iostream>

#include <fstream>

#include <string>

#include "Matrix.h"

using namespace std;

void InputDataFromFile(List<string>\* data, ifstream& file)

{

while (!file.eof())

{

string str;

getline(file, str);

data->push\_back(str);

}

}

void printInfoSchedule(List<string>\* list\_fly)

{

for (int i = 0; i < list\_fly->get\_size(); ++i)

{

cout << list\_fly->at(i) << endl;

}

}

int main() {

setlocale(LC\_ALL, "RUS");

ifstream stream("input.txt");

List<string>\* list\_fly = new List<string>();

string city\_Start;

string city\_End;

InputDataFromFile(list\_fly, stream);

cout << "Flight schedule: " << endl;

printInfoSchedule(list\_fly);

cout << "Enter the departure city" << endl;

getline(cin, city\_Start);

cout << "Enter your arrival city" << endl;

getline(cin, city\_End);

Matrix\* matrix\_floid\_uorshell = new Matrix(list\_fly);

cout << matrix\_floid\_uorshell->getResult(city\_Start, city\_End) << endl;

}

**Matrix.h:**

#pragma once

#include "List.h"

#include "Map.h"

#include <string>

class Matrix {

private:

void print\_path(int index\_start\_vertex, int index\_end\_vertex, int\*\* pre, Map<int, string>\* map\_index\_to\_name\_City, string& cur) {

if (index\_start\_vertex != index\_end\_vertex)

print\_path(index\_start\_vertex, pre[index\_start\_vertex][index\_end\_vertex], pre, map\_index\_to\_name\_City, cur);

cur += map\_index\_to\_name\_City->find(index\_end\_vertex) + " -> ";

};

void initialization(Map<string, int>\* map\_City\_name\_to\_index, Map<int, string>\* map\_index\_to\_name\_City, List<string>\* data, int& index\_city)

{

for (int i = 0; i < data->get\_size(); i++) {

string str\_cur = data->at(i);

int cur = str\_cur.find(';');

int cur1 = str\_cur.find(';', cur + 1);

string str\_name\_city1 = str\_cur.substr(0, cur);

string str\_name\_city2 = str\_cur.substr(cur + 1, cur1 - cur - 1);

str\_name\_city2.erase(0, 1);

if (!map\_City\_name\_to\_index->find\_is(str\_name\_city1)) {

map\_City\_name\_to\_index->insert(str\_name\_city1, index\_city);

map\_index\_to\_name\_City->insert(index\_city, str\_name\_city1);

index\_city++;

}

if (!map\_City\_name\_to\_index->find\_is(str\_name\_city2)) {

map\_City\_name\_to\_index->insert(str\_name\_city2, index\_city);

map\_index\_to\_name\_City->insert(index\_city, str\_name\_city2);

index\_city++;

}

}

}

void inputMatrixPath(List<string>\* data)

{

for (int i = 0; i < data->get\_size(); ++i)

{

int price\_1\_to\_2 = INF;

int price\_2\_to\_1 = INF;

string str\_cur = data->at(i);

int cur = str\_cur.find(';');

int cur1 = str\_cur.find(';', cur + 1);

int cur2 = str\_cur.find(';', cur1 + 1);

int cur3 = str\_cur.find(';', cur2 + 1);

string str\_name\_city1 = str\_cur.substr(0, cur);

string str\_name\_city2 = str\_cur.substr(cur + 1, cur1 - cur - 1);

str\_name\_city2.erase(0, 1);

if (str\_cur.substr(cur1 + 2, cur2 - 2 - cur1) != "N/A")

{

price\_1\_to\_2 = stof(str\_cur.substr(cur1 + 2, cur2 - 2 - cur1));

}

if (str\_cur.substr(cur2 + 2, cur3 - 1) != "N/A")

{

price\_2\_to\_1 = stoi(str\_cur.substr(cur2 + 2, cur3 - 2 - cur2));

}

matrix[map\_City\_name\_to\_index->find(str\_name\_city1)][map\_City\_name\_to\_index->find(str\_name\_city2)] = price\_1\_to\_2;

matrix[map\_City\_name\_to\_index->find(str\_name\_city2)][map\_City\_name\_to\_index->find(str\_name\_city1)] = price\_2\_to\_1;

}

}

double\*\* matrix;

int size\_of\_matrix;

Map<string, int>\* map\_City\_name\_to\_index;

Map<int, string>\* map\_index\_to\_name\_City;

const int INF = std::numeric\_limits<int>::max();

public:

Matrix(List<string>\* data) {

map\_City\_name\_to\_index = new Map<string, int>();

map\_index\_to\_name\_City = new Map<int, string>();

int index\_city = 0;

initialization(map\_City\_name\_to\_index, map\_index\_to\_name\_City, data, index\_city);

size\_of\_matrix = index\_city;

matrix = new double\* [size\_of\_matrix];

for (int i = 0; i < size\_of\_matrix; ++i)

{

matrix[i] = new double[size\_of\_matrix];

}

for (int i = 0; i < size\_of\_matrix; ++i)

{

for (int j = 0; j < size\_of\_matrix; ++j)

{

matrix[i][j] = INF;

}

}

inputMatrixPath(data);

}

string getResult(string& start\_City, string& end\_City) {

string cur;

while (!map\_City\_name\_to\_index->find\_is(start\_City)) {

cout << "The departure city is missing, enter it again" << endl;

cin >> start\_City;

}

while (!map\_City\_name\_to\_index->find\_is(end\_City)) {

cout << "The arrival city is missing, enter it again" << endl;

cin >> end\_City;

}

int index\_start\_vertex = map\_City\_name\_to\_index->find(start\_City);

int index\_end\_vertex = map\_City\_name\_to\_index->find(end\_City);

int\*\* pre = new int\* [size\_of\_matrix];

for (int i = 0; i < size\_of\_matrix; i++) {

pre[i] = new int[size\_of\_matrix];

for (int j = 0; j < size\_of\_matrix; j++)

pre[i][j] = i;

}

for (int k = 0; k < size\_of\_matrix; ++k)

for (int i = 0; i < size\_of\_matrix; ++i)

for (int j = 0; j < size\_of\_matrix; ++j) {

if (matrix[i][k] + matrix[k][j] < matrix[i][j]) {

matrix[i][j] = matrix[i][k] + matrix[k][j];

pre[i][j] = pre[k][j];

}

}

if (matrix[map\_City\_name\_to\_index->find(start\_City)][map\_City\_name\_to\_index->find(end\_City)] != INF) {

cur = "The best route for the price: " + to\_string(matrix[map\_City\_name\_to\_index->find(start\_City)][map\_City\_name\_to\_index->find(end\_City)]) + '\n' + "Route: ";

print\_path(index\_start\_vertex, index\_end\_vertex, pre, map\_index\_to\_name\_City, cur);

cur.erase(cur.size() - 3);

}

else {

cur = "This route can't be built, try waiting for the flight schedule for tomorrow!";

}

return cur;

}

};

**List.h:**

#pragma once

#include<iostream>

using namespace std;

template<typename TValue>

class List

{

private:

class Node {

public:

Node(TValue data = TValue(), Node\* Next = NULL) {

this->data = data;

this->Next = Next;

}

Node\* Next;

TValue data;

};

Node\* head;

Node\* tail;

size\_t size;

public:

void push\_back(TValue obj) {

if (head != NULL) {

this->tail->Next = new Node(obj);

tail = tail->Next;

}

else {

this->head = new Node(obj);

this->tail = this->head;

}

size++;

}

void push\_front(TValue obj) {

if (head != NULL) {

Node\* current = new Node;

current->data = obj;

current->Next = this->head;

this->head = current;

}

else {

this->head = new Node(obj);

}

this->size++;

}

void pop\_back() {

if (head != NULL) {

Node\* current = head;

while (current->Next != tail)

current = current->Next;

delete tail;

tail = current;

tail->Next = NULL;

size--;

}

else throw std::out\_of\_range("out\_of\_range");

}

void pop\_front() {

if (head != NULL) {

Node\* current = head;

head = head->Next;

delete current;

size--;

}

else throw std::out\_of\_range("out\_of\_range");

}

void insert(TValue obj, size\_t index) {

if (index >= 0 && this->size > index) {

if (this->head != NULL) {

if (index == 0)

this->push\_front(obj);

else

if (index == this->size - 1)

this->push\_back(obj);

else

{

Node\* current = new Node;

Node\* current1 = head;

for (int i = 0; i < index - 1; i++) {

current1 = current1->Next;

}

current->data = obj;

current->Next = current1->Next;

current1->Next = current;

size++;

}

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

TValue at(size\_t index) {

if (this->head != NULL && index >= 0 && index <= this->size - 1) {

if (index == 0)

return this->head->data;

else

if (index == this->size - 1)

return this->tail->data;

else

{

Node\* current = head;

for (int i = 0; i < index; i++) {

current = current->Next;

}

return current->data;

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

void remove(int index) { // удаление элемента по индексу

if (head != NULL && index >= 0 && index <= size - 1) {

if (index == 0) this->pop\_front();

else

if (index == this->size - 1) this->pop\_back();

else

if (index != 0) {

Node\* current = head;

for (int i = 0; i < index - 1; i++) {//переходим на предэлемент

current = current->Next;

}

Node\* current1 = current->Next;

current->Next = current->Next->Next;

delete current1;

size--;

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

size\_t get\_size() { // получение размера списка

return size;

}

void print\_to\_console() { // вывод элементов списка в консоль через разделитель

if (this->head != NULL) {

Node\* current = head;

for (int i = 0; i < size; i++) {

cout << current->data << ' ';

current = current->Next;

}

}

}

void clear() { // удаление всех элементов списка

if (head != NULL) {

Node\* current = head;

while (head != NULL) {

current = current->Next;

delete head;

head = current;

}

size = 0;

}

}

void set(size\_t index, TValue obj) // замена элемента по индексу на передаваемый элемент

{

if (this->head != NULL && this->get\_size() >= index && index >= 0) {

Node\* current = head;

for (int i = 0; i < index; i++) {

current = current->Next;

}

current->data = obj;

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

bool isEmpty() { // проверка на пустоту списка

return (bool)(head);

}

void reverse() { // меняет порядок элементов в списке

int Counter = size;

Node\* HeadCur = NULL;

Node\* TailCur = NULL;

for (int j = 0; j < size; j++) {

if (HeadCur != NULL) {

if (head != NULL && head->Next == NULL) {

TailCur->Next = head;

TailCur = head;

head = NULL;

}

else {

Node\* cur = head;

for (int i = 0; i < Counter - 2; i++)

cur = cur->Next;

TailCur->Next = cur->Next;

TailCur = cur->Next;

cur->Next = NULL;

tail = cur;

Counter--;

}

}

else {

HeadCur = tail;

TailCur = tail;

Node\* cur = head;

for (int i = 0; i < size - 2; i++)

cur = cur->Next;

tail = cur;

tail->Next = NULL;

Counter--;

}

}

head = HeadCur;

tail = TailCur;

}

List(Node\* head = NULL, Node\* tail = NULL, size\_t size = 0) :head(head), tail(tail), size(size) {}

~List() {

if (head != NULL) {

this->clear();

}

};

};

**Map.h:**

#pragma once

#include "List.h"

using namespace std;

enum Color

{

BLACK, RED

};

template<typename TKey, typename TValue>

class Map {

public:

class Node

{

public:

Node(bool color = RED, TKey key = TKey(), Node\* parent = NULL, Node\* left = NULL, Node\* right = NULL, TValue value = TValue()) :color(color), key(key), parent(parent), left(left), right(right), value(value) {}

TKey key;

TValue value;

bool color;

Node\* parent;

Node\* left;

Node\* right;

};

~Map() {

if (this->Top != NULL)

this->clear();

Top = NULL;

delete TNULL;

TNULL = NULL;

}

Map(Node\* Top = NULL, Node\* TNULL = new Node(0)) :Top(TNULL), TNULL(TNULL) {}

void printTree()

{

if (Top)

{

print\_Helper(this->Top, "", true);

}

else throw std::out\_of\_range("Tree is empty!");

}

void insert(TKey key, TValue value)

{

if (this->Top != TNULL) {

Node\* node = NULL;

Node\* parent = NULL;

/\* Search leaf for new element \*/

for (node = this->Top; node != TNULL; )

{

parent = node;

if (key < node->key)

node = node->left;

else if (key > node->key)

node = node->right;

else if (key == node->key)

throw std::out\_of\_range("key is repeated");

}

node = new Node(RED, key, TNULL, TNULL, TNULL, value);

node->parent = parent;

if (parent != TNULL) {

if (key < parent->key)

parent->left = node;

else

parent->right = node;

}

rbtree\_fixup\_add(node);

}

else {

this->Top = new Node(BLACK, key, TNULL, TNULL, TNULL, value);

}

}

List<TKey>\* get\_keys() {

List<TKey>\* list = new List<TKey>();

this->ListKeyOrValue(1, list);

return list;

}

List<TValue>\* get\_values() {

List<TValue>\* list = new List<TValue>();

this->ListKeyOrValue(2, list);

return list;

}

TValue find(TKey key) {

Node\* node = Top;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node->value;

else

throw std::out\_of\_range("Key is missing");

}

bool find\_is(TKey key) {

Node\* node = Top;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return true;

else

return false;

}

void remove(TKey key) {

this->deleteNodeHelper(this->find\_key(key));

}

void clear() {

this->clear\_tree(this->Top);

this->Top = NULL;

}

private:

Node\* Top;

Node\* TNULL;

void deleteNodeHelper(Node\* find\_node)

{

Node\* node\_with\_fix, \* cur\_for\_change;

cur\_for\_change = find\_node;

bool cur\_for\_change\_original\_color = cur\_for\_change->color;

if (find\_node->left == TNULL)

{

node\_with\_fix = find\_node->right;

Transplant(find\_node, find\_node->right);

}

else if (find\_node->right == TNULL)

{

node\_with\_fix = find\_node->left;

Transplant(find\_node, find\_node->left);

}

else

{

cur\_for\_change = minimum(find\_node->right);

cur\_for\_change\_original\_color = cur\_for\_change->color;

node\_with\_fix = cur\_for\_change->right;

if (cur\_for\_change->parent == find\_node)

{

node\_with\_fix->parent = cur\_for\_change;

}

else

{

Transplant(cur\_for\_change, cur\_for\_change->right);

cur\_for\_change->right = find\_node->right;

cur\_for\_change->right->parent = cur\_for\_change;

}

Transplant(find\_node, cur\_for\_change);

cur\_for\_change->left = find\_node->left;

cur\_for\_change->left->parent = cur\_for\_change;

cur\_for\_change->color = find\_node->color;

}

delete find\_node;

if (cur\_for\_change\_original\_color == BLACK)

{

this->rbtree\_fixup\_add(node\_with\_fix);

}

}

//swap links(parent and other) for rotate

void Transplant(Node\* cur, Node\* cur1)

{

if (cur->parent == TNULL)

{

Top = cur1;

}

else if (cur == cur->parent->left)

{

cur->parent->left = cur1;

}

else

{

cur->parent->right = cur1;

}

cur1->parent = cur->parent;

}

void clear\_tree(Node\* tree) {

if (tree != TNULL) {

clear\_tree(tree->left);

clear\_tree(tree->right);

delete tree;

}

}

Node\* minimum(Node\* node)

{

while (node->left != TNULL)

{

node = node->left;

}

return node;

}

Node\* maximum(Node\* node)

{

while (node->right != TNULL)

{

node = node->right;

}

return node;

}

Node\* grandparent(Node\* cur)

{

if ((cur != TNULL) && (cur->parent != TNULL))

return cur->parent->parent;

else

return TNULL;

}

Node\* uncle(Node\* cur)

{

Node\* cur1 = grandparent(cur);

if (cur1 == TNULL)

return TNULL;

if (cur->parent == cur1->left)

return cur1->right;

else

return cur1->left;

}

Node\* sibling(Node\* n)

{

if (n == n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

Node\* find\_key(TKey key) {

Node\* node = this->Top;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node;

else

throw std::out\_of\_range("Key is missing");

}

void print\_Helper(Node\* root, string indent, bool last)

{

if (root != TNULL)

{

cout << indent;

if (last)

{

cout << "R----";

indent += " ";

}

else

{

cout << "L----";

indent += "| ";

}

string sColor = !root->color ? "BLACK" : "RED";

cout << root->key << "(" << sColor << ")" << endl;

print\_Helper(root->left, indent, false);

print\_Helper(root->right, indent, true);

}

}

void ListKeyOrValue(int mode, List<TKey>\* list) {

if (this->Top != TNULL)

this->KeyOrValue(Top, list, mode);

else

throw std::out\_of\_range("Tree empty!");

}

void KeyOrValue(Node\* tree, List<TKey>\* list, int mode) {

if (tree != TNULL) {

KeyOrValue(tree->left, list, mode);

if (mode == 1)

list->push\_back(tree->key);

else

list->push\_back(tree->value);

KeyOrValue(tree->right, list, mode);

}

}

void rbtree\_fixup\_add(Node\* node)

{

Node\* uncle;

/\* Current node is RED \*/

while (node != this->Top && node->parent->color == RED)//

{

/\* node in left tree of grandfather \*/

if (node->parent == this->grandparent(node)->left)//

{

/\* node in left tree of grandfather \*/

uncle = this->uncle(node);

if (uncle->color == RED) {

/\* Case 1 - uncle is RED \*/

node->parent->color = BLACK;

uncle->color = BLACK;

this->grandparent(node)->color = RED;

node = this->grandparent(node);

}

else {

/\* Cases 2 & 3 - uncle is BLACK \*/

if (node == node->parent->right) {

/\*Reduce case 2 to case 3 \*/

node = node->parent;

this->left\_rotate(node);

}

/\* Case 3 \*/

node->parent->color = BLACK;

this->grandparent(node)->color = RED;

this->right\_rotate(this->grandparent(node));

}

}

else {

/\* Node in right tree of grandfather \*/

uncle = this->uncle(node);

if (uncle->color == RED) {

/\* Uncle is RED \*/

node->parent->color = BLACK;

uncle->color = BLACK;

this->grandparent(node)->color = RED;

node = this->grandparent(node);

}

else {

/\* Uncle is BLACK \*/

if (node == node->parent->left) {

node = node->parent;

this->right\_rotate(node);

}

node->parent->color = BLACK;

this->grandparent(node)->color = RED;

this->left\_rotate(this->grandparent(node));

}

}

}

this->Top->color = BLACK;

}

void left\_rotate(Node\* node)

{

Node\* right = node->right;

/\* Create node->right link \*/

node->right = right->left;

if (right->left != TNULL)

right->left->parent = node;

/\* Create right->parent link \*/

if (right != TNULL)

right->parent = node->parent;

if (node->parent != TNULL) {

if (node == node->parent->left)

node->parent->left = right;

else

node->parent->right = right;

}

else {

this->Top = right;

}

right->left = node;

if (node != TNULL)

node->parent = right;

}

void right\_rotate(Node\* node)

{

Node\* left = node->left;

/\* Create node->left link \*/

node->left = left->right;

if (left->right != TNULL)

left->right->parent = node;

/\* Create left->parent link \*/

if (left != TNULL)

left->parent = node->parent;

if (node->parent != TNULL) {

if (node == node->parent->right)

node->parent->right = left;

else

node->parent->left = left;

}

else {

this->Top = left;

}

left->right = node;

if (node != TNULL)

node->parent = left;

}

};

**TestProject.cpp:**

#include "CppUnitTest.h"

#include <fstream>

#include<string>

#include"../Laba3/Matrix.h"

#include"../Laba3/Laba3.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTestForAlgorithmFloydUorshell

{

TEST\_CLASS(UnitTestForAlgorithmFloydUorshell)

{

public:

TEST\_METHOD(Test\_Path\_is\_avaible)

{

ifstream stream("C:\\Users\ASUS\\Desktop\\Лабораторная работа №3\\Laba3\\TestProject\\example1.txt");

List<string>\* list\_fly = new List<string>();

string city\_Start = "Vladivostok";

string city\_End = "Moscow";

InputDataFromFile(list\_fly, stream);

Matrix\* matrix\_floid\_uorshell = new Matrix(list\_fly);

string excepted = "The best route for the price: 30.000000\nRoute: Vladivostok -> Saint Petersburg -> Moscow ";

Assert::AreEqual(matrix\_floid\_uorshell->getResult(city\_Start, city\_End), excepted);

}

TEST\_METHOD(Test\_Path\_is\_not\_avaible)

{

ifstream stream("C:\\Users\\ASUS\\Desktop\\Лабораторная работа №3\\Laba3\\TestProject\\example2.txt");

List<string>\* list\_fly = new List<string>();

string city\_Start = "Sochi";

string city\_End = "Saint Petersburg";

InputDataFromFile(list\_fly, stream);

Matrix\* matrix\_floid\_uorshell = new Matrix(list\_fly);

string excepted = "This route can't be built, try waiting for the flight schedule for tomorrow!";

Assert::AreEqual(matrix\_floid\_uorshell->getResult(city\_Start, city\_End), excepted);

}

};

}